Problem 1

Proofs make use of the properties/rules from Lecture 06 - slides (7, 15, 16, 17).

- $f(x, y, z) = 3x + e^{y+z} - \min\{-x^2, \quad \log(y)\}$, on $D = (\text{-}100, 100) \times (1, 50) \times (10, 20)$ is convex.

  - $3x$ is convex by Rule 1.
  - e^{y+z} is convex by Rule 1 and 2.
  - x^2 is convex (i.e. it's second order derivative 2 is non-negative on D, thus making $-x^2$ concave. $\log(y)$ is concave. $\min\{-x^2, \quad \log(y)\}$ returns $-x^2$ by construction of D, making $-\min\{-x^2, \quad \log(y)\}$ convex.

  By Rule 1, the sum of these convex functions is also convex. It follows, $f(x, y, z)$ is convex.

- $f(x, y) = yx^3 - 2yx^2 + y + 4$, $D = (\text{-}10, 10) \times (\text{-}10, 10)$ is not convex because x^3 its second order derivative is not non-negative on the interval D. (cf. Lecture 06 - slide 15)

- $f(x) = \log(x) + x^3$ and $D = (1, \infty)$ is not convex because $\log(x)$ is concave by definition.

- $f(x) = \text{-}\min(2\log(2x), \text{-}x^2 + 4x \quad \text{-}32)$, $D = \mathbb{R}^+$ is convex, because $\text{-}x^2 + 4x \quad \text{-}32$ is concave quadratic function which reaches its global maximum of 28 at $x = 2$ (e.g. use high-school to determine maximum). On the other hand $\log(2x)$ is non-negative, meaning that $\min(\cdot, \cdot)$ gives by construction the quadratic function. Finally, the negative sign makes $f(x)$ convex.

Problem 2

Let $x, y \in \mathbb{R}^d, \alpha \in [0, 1]$.

$$
\begin{aligned}
h(\alpha x + (1-\alpha)y) &= f_1(\alpha x + (1-\alpha)y) + f_2(\alpha x + (1-\alpha)y) && \text{(by def.)} \\
&\leq \alpha f_1(x) + (1-\alpha) f_1(y) + \alpha f_2(x) + (1-\alpha) f_2(y) && \text{(by convexity of } f_1, f_2\text{)} \\
&= \alpha f_1(x) + \alpha f_2(x) + (1-\alpha) f_1(y) + (1-\alpha) f_2(y) \\
&= \alpha h(x) + (1-\alpha) h(y)
\end{aligned}
$$

Problem 3

Proof by counterexample. Let $f_1(x) = x^2$ and $f_2(x) = x$. Both functions are convex (by Exercise 1). But, $g(x) = f_1(x) \cdot f_2(x) = x^3$ is not convex on $\mathbb{R}$ because its second derivative is not non-negative over this interval.

Problem 4

Proof by contradiction. Assume $\nabla_\theta f(\theta^*) = 0$ and θ^* is not a global minimum. By first order convexity of f we have $\forall x, y$ it must hold $f(y) \geq f(x) + (y - x)^\intercal \nabla_x f(x)$. Plug $x = \theta^*$ we get $f(y) \geq f(\theta^*)$. But θ^* was not global minimum, so we get contradiction. Thus θ^* must be the global minimum.

Problem 5

See notebook below.

06_optimization_logistic_regression

November 30, 2017

1 Programming assignment 6: Optimization: Logistic regression

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        from sklearn.datasets import load_breast_cancer
        from sklearn.model_selection import train_test_split
        from sklearn.metrics import accuracy_score, f1_score
```

1.1 Your task

In this notebook code skeleton for performing logistic regression with gradient descent is given. Your task is to complete the functions where required. You are only allowed to use built-in Python functions, as well as any `numpy` functions. No other libraries / imports are allowed.

For numerical reasons, we actually minimize the following loss function

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} NLL(\mathbf{w}) + \lambda ||\mathbf{w}||_2^2$$

where $NLL(\mathbf{w})$ is the negative log-likelihood function, as defined in the lecture (Eq. 33)

1.2 Load and preprocess the data

In this assignment we will work with the UCI ML Breast Cancer Wisconsin (Diagnostic) dataset https://goo.gl/U2Uwz2.

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. There are 212 malignant examples and 357 benign examples.

```
In [2]: X, y = load_breast_cancer(return_X_y=True)

        # Add a vector of ones to the data matrix to absorb the bias term
        X = np.hstack([np.ones([X.shape[0], 1]), X])

        # Set the random seed so that we have reproducible experiments
        np.random.seed(123)
```

```
        # Split into train and test
        test_size = 0.3
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)
```

1.3 Task 1: Implement the sigmoid function

```
In [3]: def sigmoid(t):
            """
            Applies the sigmoid function elementwise to the input data.

            Parameters
            ----------
            t : array, arbitrary shape
                Input data.

            Returns
            -------
            t_sigmoid : array, arbitrary shape.
                Data after applying the sigmoid function.
            """
            return 1 / (1 + np.exp(-t))
```

1.4 Task 2: Implement the negative log likelihood

As defined in Eq. 33

```
In [3]: def negative_log_likelihood(X, y, w, eps=1e-15):
            """
            Negative Log Likelihood of the Logistic Regression.

            Parameters
            ----------
            X : array, shape [N, D]
                (Augmented) feature matrix.
            y : array, shape [N]
                Classification targets.
            w : array, shape [D]
                Regression coefficients (w[0] is the bias term).

            Returns
            -------
            nll : float
                The negative log likelihood.
            """
            loss = -np.sum(y * np.log(sigmoid(np.dot(X, w)) + eps) +
                                                (1 - y) *
                                                np.log(1 - sigmoid(np.dot(X, w)) + eps),
                                                axis=0)
```

```
    return loss
```

1.4.1 Computing the loss function $\mathcal{L}(\mathbf{w})$ (nothing to do here)

```
In [5]: def compute_loss(X, y, w, lmbda):
            """
            Negative Log Likelihood of the Logistic Regression.

            Parameters
            ----------
            X : array, shape [N, D]
                (Augmented) feature matrix.
            y : array, shape [N]
                Classification targets.
            w : array, shape [D]
                Regression coefficients (w[0] is the bias term).
            lmbda : float
                L2 regularization strength.

            Returns
            -------
            loss : float
                Loss of the regularized logistic regression model.
            """
            # The bias term w[0] is not regularized by convention
            return negative_log_likelihood(X, y, w) / len(y) + lmbda * np.linalg.norm(w[1:])
```

1.5 Task 3: Implement the gradient $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$

Make sure that you compute the gradient of the loss function $\mathcal{L}(\mathbf{w})$ (not simply the NLL!)

```
In [2]: def get_gradient(X, y, w, mini_batch_indices, lmbda):
            """
            Calculates the gradient (full or mini-batch) of the negative log likelilhood w.r.t

            Parameters
            ----------
            X : array, shape [N, D]
                (Augmented) feature matrix.
            y : array, shape [N]
                Classification targets.
            w : array, shape [D]
                Regression coefficients (w[0] is the bias term).
            mini_batch_indices: array, shape [mini_batch_size]
                The indices of the data points to be included in the (stochastic) calculation
                This includes the full batch gradient as well, if mini_batch_indices = np.aran
            lmbda: float
```

```
        Regularization strentgh. lmbda = 0 means having no regularization.

    Returns
    -------
    dw : array, shape [D]
        Gradient w.r.t. w.
    """
    grad_data = np.zeros_like(w)
    grad_data = np.sum(np.expand_dims((-y[mini_batch_indices] *
                                       (1 - sigmoid(np.dot(X[mini_batch_indices, :],
                                                           w)))),
                                      axis=1) * X[mini_batch_indices, :] +
                       np.expand_dims(((1 - y[mini_batch_indices]) *
                                       (sigmoid(np.dot(X[mini_batch_indices, :], w))))
                                      axis=1) * X[mini_batch_indices, :], axis=0)

    grad_data = grad_data * 1 / len(mini_batch_indices)
    grad_reg = 2 * lmbda * w
    grad = grad_data + grad_reg

    return grad
```

1.5.1 Train the logistic regression model (nothing to do here)

```
In [20]: def logistic_regression(X, y, num_steps, learning_rate, mini_batch_size, lmbda, verbos
             """
             Performs logistic regression with (stochastic) gradient descent.

             Parameters
             ----------
             X : array, shape [N, D]
                 (Augmented) feature matrix.
             y : array, shape [N]
                 Classification targets.
             num_steps : int
                 Number of steps of gradient descent to perform.
             learning_rate: float
                 The learning rate to use when updating the parameters w.
             mini_batch_size: int
                 The number of examples in each mini-batch.
                 If mini_batch_size=n_train we perform full batch gradient descent.
             lmbda: float
                 Regularization strentgh. lmbda = 0 means having no regularization.
             verbose : bool
                 Whether to print the loss during optimization.

             Returns
             -------
```

```
    w : array, shape [D]
        Optimal regression coefficients (w[0] is the bias term).
    trace: list
        Trace of the loss function after each step of gradient descent.
    """

    trace = [] # saves the value of loss every 50 iterations to be able to plot it la
    n_train = X.shape[0] # number of training instances

    w = np.zeros(X.shape[1]) # initialize the parameters to zeros

    # run gradient descent for a given number of steps
    for step in range(num_steps):
        permuted_idx = np.random.permutation(n_train) # shuffle the data

        # go over each mini-batch and update the paramters
        # if mini_batch_size = n_train we perform full batch GD and this loop runs on
        for idx in range(0, n_train, mini_batch_size):
            # get the random indices to be included in the mini batch
            mini_batch_indices = permuted_idx[idx:idx+mini_batch_size]
            gradient = get_gradient(X, y, w, mini_batch_indices, lmbda)

            # update the parameters
            w = w - learning_rate * gradient

        # calculate and save the current loss value every 50 iterations
        if step % 2000 == 0:
            loss = compute_loss(X, y, w, lmbda)
            trace.append(loss)
            # print loss to monitor the progress
            if verbose:
                print('Step {0}, loss = {1:.4f}'.format(step, loss))
    return w, trace
```

1.6 Task 4: Implement the function to obtain the predictions

In [21]:
```
def predict(X, w):
    """
    Parameters
    ----------
    X : array, shape [N_test, D]
        (Augmented) feature matrix.
    w : array, shape [D]
        Regression coefficients (w[0] is the bias term).

    Returns
    -------
    y_pred : array, shape [N_test]
```

```
        A binary array of predictions.
    """
    return np.where(np.dot(X, w) > 0, 1, 0)
```

1.6.1 Full batch gradient descent

```
In [22]: # Change this to True if you want to see loss values over iterations.
         verbose = True
```

```
In [23]: n_train = X_train.shape[0]
         w_full, trace_full = logistic_regression(X_train,
                                                  y_train,
                                                  num_steps=8000,
                                                  learning_rate=1e-5,
                                                  mini_batch_size=n_train,
                                                  lmbda=0.1,
                                                  verbose=verbose)
```

```
Step 0, loss = 0.7428
Step 2000, loss = 0.2420
Step 4000, loss = 0.2272
Step 6000, loss = 0.2224
```

```
In [32]: n_train = X_train.shape[0]
         w_minibatch, trace_minibatch = logistic_regression(X_train,
                                                            y_train,
                                                            num_steps=8000,
                                                            learning_rate=1e-5,
                                                            mini_batch_size=50,
                                                            lmbda=0.1,
                                                            verbose=verbose)
```

```
Step 0, loss = 0.6347
Step 2000, loss = 0.2244
Step 4000, loss = 0.2103
Step 6000, loss = 0.2085
```

Our reference solution produces, but don't worry if yours is not exactly the same.

```
Full batch: accuracy: 0.9240, f1_score: 0.9384
Mini-batch: accuracy: 0.9415, f1_score: 0.9533
```

```
In [33]: y_pred_full = predict(X_test, w_full)
         y_pred_minibatch = predict(X_test, w_minibatch)

         print('Full batch: accuracy: {:.4f}, f1_score: {:.4f}'
               .format(accuracy_score(y_test, y_pred_full), f1_score(y_test, y_pred_full)))
         print('Mini-batch: accuracy: {:.4f}, f1_score: {:.4f}'
               .format(accuracy_score(y_test, y_pred_minibatch), f1_score(y_test, y_pred_miniba
```

```
Full batch: accuracy: 0.9240, f1_score: 0.9384
Mini-batch: accuracy: 0.9415, f1_score: 0.9528
```

```
In [34]: plt.figure(figsize=[15, 10])
         plt.plot(trace_full, label='Full batch')
         plt.plot(trace_minibatch, label='Mini-batch')
         plt.xlabel('Iterations * 50')
         plt.ylabel('Loss $\mathcal{L}(\mathbf{w})$')
         plt.legend()
         plt.show()
```